Token-based Atomic Broadcast using Unreliable Failure Detectors

Richard Ekwall[†] nilsrichard.ekwall@epfl.ch

André Schiper[†] andre.schiper@epfl.ch

Péter Urbán[‡] urban@jaist.ac.jp

[†]École Polytechnique Fédérale de Lausanne (EPFL), LSR–IIF–I&C, CH-1015 Lausanne, Switzerland
phone:+41-21-693 6463, fax:+41-21-693 6770

[‡]Japan Advanced Institute of Science and Technology (JAIST), Ishikawa 923-1292, Japan

Technical Report IC/2004/40

Abstract

Many atomic broadcast algorithms have been published in the last twenty years. Token-based algorithms represent a large class of these algorithms. Interestingly, all the token-based atomic broadcast algorithms rely on a *group membership* service, i.e., none of them uses unreliable failure detectors directly. The paper presents the first token-based atomic broadcast algorithm that uses an unreliable failure detector – the new failure detector denoted by $\mathcal{R}$ – instead of a group membership service. The failure detector $\mathcal{R}$ is compared with $\diamondsuit\mathcal{P}$ and $\diamondsuit\mathcal{S}$. In order to make it easier to understand the atomic broadcast algorithm, the paper derives the atomic broadcast algorithm from a token-based consensus algorithm that also uses the failure detector $\mathcal{R}$.

Keywords: Atomic Broadcast, Consensus, Token, Failure Detector

Contact author : Richard Ekwall

1. INTRODUCTION

1.1 Context

Atomic broadcast (or total order broadcast) is an important abstraction in fault-tolerant distributed computing. Atomic broadcast ensures that messages broadcast by different processes are delivered by all destination processes in the same order [11]. Many atomic broadcast algorithms have been published in the last twenty years. These algorithms can be classified according to the mechanism used for message ordering [8]. *Token circulation* is one important ordering mechanism. In these algorithms, a token circulates among the processes, and the token holder has the privilege to order messages that have been broadcast. Additionally, sometimes only the token holder is

Research supported by OFES under contract number 01.0537-1 as part of the IST REMUNE project (number 2001-65002), the Japan Society for the Promotion of Science, a Grant-in-Aid for JSPS Fellows from the Japanese Ministry of Education, Culture, Sports, Science and Technology, and the Swiss National Science Foundation.

allowed to broadcast messages. However, the ordering mechanism is not the only key mechanism of an atomic broadcast algorithm. The mechanism used to tolerate failures is another important characteristic of these algorithms. If we consider asynchronous systems with crash failures, the two most widely used mechanisms to tolerate failures in the context of atomic broadcast algorithms are (i) *unreliable failure detectors* [3] and (ii) *group membership* [5]. For example, the atomic broadcast algorithm in [3] (together with a consensus algorithm using the failure detector $\Diamond\mathcal{S}$ [3]) falls into the first category; the atomic broadcast algorithm in [2] falls into the second category.

1.2 Group membership mechanism $vs.$ failure detector mechanism.

A group membership service provides a consistent membership information to all the members of a group [5]. Its main feature is to *remove* processes that are suspected to have crashed.[1] In contrast, an unreliable failure detector, e.g., $\Diamond\mathcal{S}$, does not provide consistent information about the failure status of processes. For example, it can tell to process p that r has crashed, while telling at the same time to process q that r is alive.

Both mechanisms can make mistakes, e.g., by incorrectly suspecting correct processes. However, the cost of a wrong failure suspicion is higher when using a group membership service than when using failure detectors. This is because the group membership service removes suspected processes from the group, a costly operation. *This removal is absolutely necessary for the atomic broadcast that relies on the membership service: the notification of the removal allows the atomic broadcast algorithm to avoid being blocked.* There is no such removal of suspected processes with a failure detector. Moreover, with a group membership service, the removal of a process is usually followed by the addition of another (or the same) process, in order to keep the same replication degree. So, with a group membership service, a wrong suspicion leads to two costly membership operations: *removal* of a process followed by the *addition* of another process.

In an environment where wrong failure suspicions are frequent,[2] algorithms based on failure detectors thus have advantages over algorithms based on a group membership service. The cost difference has been experimentally evaluated in [19] in the context of two specific (not token-based) atomic broadcast algorithm.

Atomic broadcasts algorithms based on a failure detector have another important advantage over algorithms based on group membership: *they can be used to implement the group membership service*! Indeed, since a (primary partition) group membership service orders views, it seems intuitive to solve group membership using atomic broadcast: this leads to a much simpler protocol stack than implementing atomic broadcast using group membership [14]. However, this is not possible if atomic broadcast relies on group membership.

[1] The comment applies to the so-called *primary-partition* membership [5].

[2] This typically happens if the timeouts used to suspect processes have been set to small values (i.e., in the order of the average message transmission delay), in order to reduce the time needed to detect the crash of processes.

1.3 Why token-based algorithms?

According to [20], [1], [13], token-based atomic broadcast algorithms are extremely efficient in terms of throughput, i.e., the number of messages that can be delivered per time unit. The reason is that these algorithms manage to reduce network contention by using the token (1) to avoid the ack explosion problem (which happens if each broadcast message generates one acknowledgement per receiving process), and/or (2) to perform flow control (e.g., a process is allowed to broadcast a message only when holding the token). However, none of the token-based algorithms use failure detectors: they all rely on a group membership service.[3] It is therefore interesting to try to design token-based atomic broadcast algorithms that rely on failure detectors, in order to combine the advantage of failure detectors and of token-based algorithms: good throughput (without sacrificing latency) in stable environments, but adapted to frequent wrong failure suspicions.

1.4 Contribution of the paper

The paper gives the first token-based atomic broadcast algorithm that uses unreliable failure detectors instead of group membership. This result is obtained in several steps. The paper first gives a new and more general definition for token-based algorithms (Sect. 2) and introduces a new failure detector, denoted by $\mathcal{R}$, adapted to token-based algorithms (Sect. 3). The failure detector $\mathcal{R}$ is shown to be strictly weaker than $\Diamond\mathcal{P}$, and strictly stronger than $\Diamond\mathcal{S}$. Although $\Diamond\mathcal{S}$ is strong enough to solve consensus and atomic broadcast, $\mathcal{R}$ has an interesting feature: the failure detector module of a process p_i only needs to give information about the (estimated) state of p_{i-1}. For p_{i-1}, this can be done by sending *I am alive* messages to p_i only, which is extremely cheap compared to failure detectors where each process monitors all other processes. Moreover, in the case of three processes (a frequent case in practice, tolerating one crash), our token-based algorithm works with $\Diamond\mathcal{S}$.

Section 4 concentrates on the consensus problem. First we define two classes of token-based algorithms: *token-accumulation* algorithms and *token-coordinated* algorithms. We then focus on the token-accumulation approach and give a consensus algorithm based on the failure detector $\mathcal{R}$.

An algorithm that solves atomic broadcast is presented in Section 5. The algorithm is inspired from the token-based consensus algorithm of Section 4. Note that a standard solution consists in solving atomic broadcast by reduction to consensus [3]. However, this solution is not adequate here, because the resulting algorithm is highly inefficient. Our atomic broadcast algorithm is derived from our consensus algorithm in a more complex manner. Note that we could have presented only the token-based atomic broadcast algorithm. However, the detour through the consensus algorithm makes the explanation easier to understand. Section 6 compares the performance of our new atomic broadcast algorithm with the Chandra-Toueg atomic broadcast algorithm. Related work is presented in Section 7 and Section 8 concludes the paper.

[3]The group membership mechanism does not necessarily appear explicitly in the algorithm, e.g., in [13]. It can be implemented in an ad-hoc way.

2. SYSTEM MODEL AND DEFINITIONS

We assume an asynchronous system composed of n processes taken from the set $\Pi = \{p_0, \ldots, p_{n-1}\}$, with an implicit order on the processes. The k^{th} successor of a process p_i is $p_{(i+k) mod n}$, which is, from now on, simply noted p_{i+k} for the sake of clarity. Similarly the k^{th} predecessor of p_i is simply denoted by p_{i-k}. The processes communicate by message passing over reliable channels. Processes can only fail by crashing (no Byzantine failures). A process that never crashes is said to be *correct*, otherwise it is *faulty*. At most f processes are *faulty*. The system is augmented with unreliable failure detectors [3] (see below).

2.1 The consensus problem

As in [3], we specify the (uniform) consensus problem by four properties: (1) *Termination:* Every correct process eventually decides some value, (2) *Uniform integrity:* Every process decides at most once, (3) *Uniform agreement:* No two processes (correct or not) decide a different value, and (4) *Uniform validity:* If a process decides v, then v was proposed by some process in Π.

2.2 The atomic broadcast problem

In the atomic broadcast problem, defined by the primitives *abroadcast* and *adeliver*, processes have to agree on a common total order delivery of a set of messages. Formally, we define (uniform) atomic broadcast by four properties [11]: (1) *Validity:* If a correct process p *abroadcasts* a message m, then it eventually *adelivers* m, (2) *Uniform Agreement:* If a process *adelivers* m, then all correct processes eventually *adeliver* m, (3) *Uniform Integrity:* For any message m, every process p *adelivers* m at most once and only if m was previously *abroadcast*, and (4) *Uniform Total Order:* If some process, correct or faulty, *adelivers* m before m', then every process *adelivers* m' only after it has *adelivered* m.

2.3 Token-based algorithm

In a traditional token-based algorithm, processes are organized in a logical ring and, for token transmission, communicate only with their immediate predecessor and successor (except during changes in the composition of the ring). This definition is too restrictive for failure detector-based algorithms. We define an algorithm to be *token-based* if (1) processes are organized in a logical ring, (2) each process p_i has a failure detector module FD_i that provides information only about its immediate predecessor p_{i-1} and (3) each process communicates only with its $f + 1$ predecessors and successors, where f is the number of tolerated failures.

2.4 Failure detectors

We refer below to two failure detectors introduced in [3]: $\Diamond\mathcal{P}$ and $\Diamond\mathcal{S}$. The eventual perfect failure detector $\Diamond\mathcal{P}$ is defined by the following properties: (i) *Strong Completeness:* Eventually every process that crashes is permanently

suspected by every correct process, and (ii) *Eventual Strong Accuracy:* There is a time after which correct processes are not suspected by any correct process. The $\Diamond\mathcal{S}$ failure detector is defined by (i) *Strong Completeness* and (ii) *Eventual Weak Accuracy:* There is a time after which some correct process is never suspected by any correct process.

3. FAILURE DETECTOR $\mathcal{R}$

For token-based algorithms we define a new failure detector denoted by $\mathcal{R}$ (stands for *Ring*). Given process p_i, the failure detector attached to p_i only gives information about the immediate predecessor p_{i-1}.[4] For every process p_i, $\mathcal{R}$ ensures the following properties:

(i) *Completeness*: If p_{i-1} crashes and p_i is correct, then p_{i-1} is eventually permanently suspected by p_i, and

(ii) *Accuracy*: If p_{i-1} and p_i are correct, there is a time t after which p_{i-1} is never suspected by p_i.

The relation *weaker/stronger* between failure detectors has been defined in [3]. We show that (a) $\Diamond\mathcal{P}$ is strictly stronger than $\mathcal{R}$ (denoted $\Diamond\mathcal{P} \succ \mathcal{R}$), and (b) $\mathcal{R}$ is strictly stronger than $\Diamond\mathcal{S}$ if $n \geq f(f+1)+1$ ($\mathcal{R} \succ \Diamond\mathcal{S}$).

Lemma 1: $\Diamond\mathcal{P}$ is strictly stronger than $\mathcal{R}$.

Proof : This result is easy to establish. From the definition it follows directly that $\Diamond\mathcal{P}$ is stronger or equivalent to $\mathcal{R}$, denoted by $\Diamond\mathcal{P} \succeq \mathcal{R}$. Moreover, when p_i is faulty, then $\mathcal{R}$ provides no information about p_{i-1}:[5] so $\Diamond\mathcal{P} \not\equiv \mathcal{R}$ ($\Diamond\mathcal{P}$ not equivalent to $\mathcal{R}$). Together with $\Diamond\mathcal{P} \succeq \mathcal{R}$ we have that $\Diamond\mathcal{P} \succ \mathcal{R}$. □

The relationship between $\mathcal{R}$ and $\Diamond\mathcal{S}$ is more difficult to establish. We first introduce a new failure detector $\Diamond\mathcal{S}2$ (Sect. 3.1), then show that $\Diamond\mathcal{S}2 \succ \Diamond\mathcal{S}$ (Sect. 3.2) and $\mathcal{R} \succeq \Diamond\mathcal{S}2$ if $n \geq f(f+1)+1$ (Sect. 3.3). By transitivity, we have $\mathcal{R} \succ \Diamond\mathcal{S}$ if $n \geq f(f+1)+1$.

3.1 Failure detector $\Diamond\mathcal{S}2$

For the purpose of establishing the relation between $\mathcal{R}$ and $\Diamond\mathcal{S}$ we introduce the failure detector $\Diamond\mathcal{S}2$ defined as follows:

(i) *Strong Completeness*: Eventually every process that crashes is permanently suspected by every correct process and

(ii) *Eventual "Double" Accuracy*: There is a time after which *"two"* correct processes are never suspected by any correct process.

3.2 $\Diamond\mathcal{S}2$ strictly stronger than $\Diamond\mathcal{S}$

$\Diamond\mathcal{S}$ and $\Diamond\mathcal{S}2$ differ in the accuracy property only: while $\Diamond\mathcal{S}$ requires eventually *one* correct process to be no longer suspected by all correct processes, $\Diamond\mathcal{S}2$ requires the same to hold for *two* correct processes. From the

[4]Remember the meaning of the notation p_{i-k} or p_{i+k} introduced at the beginning of Section 2.

[5]In the special case of $f = 1$, the information about p_{i-1} can be obtained indirectly, i.e., if $f = 1$, the relation between $\Diamond\mathcal{P}$ and $\mathcal{R}$ is not strict: $\Diamond\mathcal{P} \succeq \mathcal{R}$.

definition, it follows directly that $\Diamond\mathcal{S}2 \succ \Diamond\mathcal{S}$.

3.3 $\mathcal{R}$ stronger than $\Diamond\mathcal{S}2$ if $n \geq f(f+1)+1$

We show that $\mathcal{R}$ is stronger than $\Diamond\mathcal{S}2$ if $n \geq f(f+1)+1$ by giving a transformation of $\mathcal{R}$ into the failure detector $\Diamond\mathcal{S}2$.

Transformation of $\mathcal{R}$ into $\Diamond\mathcal{S}2$: Each process p_j maintains a set $correct_j$ of processes that p_j believes are correct.

(i) This set is updated as follows. Each time some process p_i changes its mind about p_{i-1} (based on $\mathcal{R}$), p_i broadcasts (using a FIFO reliable broadcast communication primitive [11]) the message $(p_{i-1}, faulty)$, respectively $(p_{i-1}, correct)$. Whenever p_j receives $(p_i, faulty)$, then p_j removes p_i from $correct_j$; whenever p_j receives $(p_i, correct)$, then p_j adds p_i to $correct_j$.

(iia) For process p_i, if $correct_i$ is equal to Π (no suspected process), the output of the transformation (the two non-suspected processes) is p_0 and p_1. All other processes are suspected.

(iib) For process p_i, if $correct_i$ is not equal to Π (at least one suspected process), the output of the transformation (the two non-suspected processes) is p_k and p_{k+1} such that k is the smallest index satisfying the following conditions: (a) p_{k-1} is not in $correct_i$, and (b) the $f-1$ immediate successors p_{k+1},...,p_{k+f-1} are in $correct_i$. Apart from p_k and p_{k+1}, all other processes are suspected.

For example, for $n = 7$, $f = 2$, and $correct_i = \{p_0, p_2, p_3, p_5\}$, the non-suspected processes for p_i are p_2 and p_3. All other processes are suspected. If $correct_i = \{p_0, p_1, p_2, p_3, p_5\}$, the non-suspected processes for p_i are p_0 and p_1 (the predecessor of p_0 is p_6, not in $correct_i$). All other processes are suspected.

Lemma 2: Consider a system with $n \geq f(f+1)+1$ processes and the failure detector $\mathcal{R}$. The above transformation guarantees that eventually all correct processes do not suspect the same two correct processes.

The proof of this lemma can be found in Appendix A.1.

The transformation of $\mathcal{R}$ into $\Diamond\mathcal{S}2$ ensures the *Eventual Double Accuracy* property if $n \geq f(f+1)+1$. Since all processes except two correct processes are suspected, the *Strong Completeness* property also holds. Consequently, if $n \geq f(f+1)+1$ we have $\mathcal{R} \succeq \Diamond\mathcal{S}2$.

4. TOKEN-BASED CONSENSUS

4.1 Two classes of algorithms

We identify two classes of token-based consensus algorithms: *token-accumulation* algorithms and *token-coordinated* algorithms. In the *token-accumulation* algorithms, each token holder votes for the proposal transported in the token. Votes are accumulated as the token circulates and once enough votes have been collected, the token holder can decide. In this class of algorithms, the only communication is related to the circulation of the token. This is not the case of *token-coordinated* algorithms. In these algorithms the token holds a proposal, but, in order to

decide, the token holder can communicate with all other processes. Algorithms based on the *rotating-coordinator paradigm* (such as the Chandra-Toueg $\Diamond\mathcal{S}$ consensus algorithm [3]) can easily be adapted to this class ([15] describes such a transformation). Token-accumulation algorithms are more genuine token-based algorithms, and the paper concentrates on this class of algorithms. Henceforth, *token-accumulation* algorithms will simply be referred to as *token-based* algorithms.

4.2 Token circulation

The token circulation is as follows. To avoid the loss of the token due to crashes, process p_i sends the token to its $f+1$ successors in the ring, i.e., to $p_{i+1}, \ldots, p_{i+f+1}$.[6] Furthermore, when awaiting the token, process p_i waits to get the token from p_{i-1}, unless it suspects p_{i-1}. If p_i suspects p_{i-1}, it accepts the token from any of its predecessors (see Procedure 1).

Procedure 1 Receive token (code of process p_i)

1: wait until received token from p_{i-1} or suspected(p_{i-1})
2: **if** token not received **then** {*accept from anyone*}
3: wait until received token from $p \in \{p_{i-f-1}, ..., p_{i-1}\}$
4: **end if**

4.3 Token-based consensus algorithm

4.3.1 Basic idea

Consensus is achieved by passing a token between the different processes. The token contains information regarding the current proposal (or the decision once it has been taken). The token is passed between the processes on a logical ring p_0, p_1,...,p_{n-1}. Each token holder "votes" for the proposal in the token and then sends it to its neighbors. As soon as a sufficient number of token holders have voted for some proposal v, then v is decided. The decision is then propagated as the token circulates along the ring.

4.3.2 Naive algorithm

We start by presenting a naive algorithm that illustrates both the basic idea behind our algorithm and its difficulty. Let the token carry an $estimate$ value (denoted by $token.estimate$) and the number of votes for this estimate (denoted $token.votes$). Let each process p_i, upon receiving the token, blindly add its vote to the proposal (see Procedure 2). Obviously, this naive algorithm does not work: it would solve consensus in an asynchronous system, in contradiction with the FLP impossibility result [10].

[6] The token should be seen as a *logical* token. Multiple backup copies circulate in the ring, but they are discarded by the algorithm if no suspicion occurs. Henceforth, the *logical* token will simply be referred to as "the token".

Procedure 2 Token handling by p_i (option 1)

$p_i.estimate \leftarrow token.estimate$
$token.votes \leftarrow token.votes + 1$
if $token.votes \geq f + 1$ **then**
  decide($token.estimate$)
end if
send token to $p_{i+1}, \ldots, p_{i+f+1}$

4.3.3 Overview of the token-based consensus algorithm

As just shown, a token-based algorithm cannot blindly increase the votes accumulated. We slightly change the above behavior. The processes need one additional information: the *gap* in the circulation of the token. When a process p_i receives the token from process $sender \equiv p_j$, the *gap* is $i - j - 1$, denoted by $gap(sender \rightarrow p_i)$. We have $gap(sender \rightarrow p_i) = 0$ only if the token is received from the immediate predecessor. Upon receiving the token, a process does the following (see Procedure 3):

As long as there is no gap in the token circulation $token.votes$ is incremented by the receiver p_i. If at that point $token.votes$ is greater than the vote threshold $f + 1$, p_i decides on the estimate of the token. The decision is then propagated with the token.

Procedure 3 Token handling by p_i (option 2)

if ($gap(sender \rightarrow p_i) \neq 0$) **then**
  $token.votes \leftarrow 0$ {*reset token*}
end if
$p_i.estimate \leftarrow token.estimate$
$token.votes \leftarrow token.votes + 1$
if $token.votes \geq f + 1$ **then**
  decide($token.estimate$)
end if
send token to $p_{i+1}, \ldots, p_{i+f+1}$

4.3.4 Conditions for agreement vs. termination

In the above algorithm, where votes are reset as soon as a gap in the token circulation is detected, Agreement holds if the vote threshold is greater or equal than $f + 1$. Termination additionally requires the failure detector $\mathcal{R}$ and that there be at least $n \geq (f + 1)f + 1$ processes in the system.

Remark: The condition $gap(sender \rightarrow p_i) = 0$ is not a necessary condition for Agreement in a token-based consensus algorithm. In [9], we present such an algorithm, parametrized with $gapThreshold$ (the number of gaps in the token circulation before resetting the vote counter) and $voteThreshold$ (the number of votes required to decide). Agreement holds if $voteThreshold \geq (gapThreshold + 1)f + 1$. However Termination still requires $gapThreshold = 0$ (in addition to $n \geq (f + 1)f + 1$ and $\mathcal{R}$).

4.3.5 Details of the algorithm

The token contains the following fields: $round$ (round number), $estimate$, $votes$ (accumulated votes for the $estimate$ value) and $decision$ (a boolean indicating if $estimate$ is the decision).

Procedure 4 Consensus: Initialization

1: $\forall p_i,\ i \in [0, n-1]$:
2: $estimate_i \leftarrow v_i$; $decided_i \leftarrow false$; $round_i \leftarrow 0$

3: p_0 : {*send token*}
4: send(0,v_0,1,false) to $\{p_1,\ldots,p_{f+1}\}$

5: $\forall p_i,\ i \in [n-f, n-1]$: {*send "dummy" token*}
6: send(-1,$\perp$,0,false) to $\{p_1, \ldots, p_{i+f+1}\}$

Procedure 5 Token-accumulation consensus: token handling by p_i

1: **loop**
2: token $\leftarrow$ receive-token($round_i$) {*see Proc. 1*}

3: **if** $token.estimate = \perp$ **then** {*use initial value*}
4: $token.estimate \leftarrow estimate_i$
5: **end if**

6: **if** not $decided_i$ **then**
7: $estimate_i \leftarrow token.estimate$
8: **if** ($gap(sender \rightarrow p_i) = 0$) **then**
9: $votes_i \leftarrow token.votes + 1$ {*add vote*}
10: **else**
11: $votes_i \leftarrow 1$; {*reset votes*}
12: **end if**
13: **if** $(votes_i \geq f + 1)$ **or** $token.decision$ **then**
14: decide($estimate_i$); $decided_i \leftarrow true$
15: **end if**
16: **end if**
17: token $\leftarrow (round_i, estimate_i, votes_i, decided_i)$
18: send token to $\{p_{i+1},\ldots,p_{i+f+1}\}$
19: $round_i \leftarrow round_i + 1$
20: **end loop**

21: **upon** reception of token s.t.
$token.round < round_i$ **do**
22: **if** $token.decision$ and (not $decided_i$) **then**
23: $estimate_i \leftarrow token.estimate$
24: $decide(estimate_i)$; $decided_i \leftarrow true$
25: **end if**
26: **end upon**

The initialization code is given by Procedure 4. Lines 5-6 show the *dummy* token sent to prevent blocking in the

case processes $p_0, \ldots, p_{f-1}$ are initially crashed. A dummy token has $round = -1$, $estimate = \perp$ and $votes = 0$, and is sent only to processes p_1, ..., p_f.

The token handling code is given by Procedure 5. At line 2, process p_i starts by receiving the token (see Procedure 1) for the expected $round_i$.[7] If no value is transported by the token ($dummy$ initialization token), p_i replaces $token.estimate$ by its own estimate (lines 3-5). If p_i has not yet decided, then p_i starts by updating its estimate (line 7). If there was no gap in the token circulation, then the votes are incremented (line 9). Otherwise, the votes are reset to 1 (line 11), which starts a new sequence of vote accumulation. At line 13, process p_i checks whether there are enough votes for a decision to be taken. If so, p_i decides (line 14). Finally, the token with the updated fields is sent to the $f + 1$ successors (line 18), and process p_i increments $round_i$ (line 19).

Lines 1-20 ensure that at least one correct process eventually decides. However, if $f > 1$, this does not ensure that all correct processes eventually decide. Consider the following example: p_i is the first process to decide, p_{i+1} is faulty. In this case, p_{i+2} may always receive the token from p_{i-1}, a token that does not carry a decision; p_i might be the only process to ever decide. Lines 21-26 ensure that every correct process eventually decides. The token received at line 2, for $round_i$, follows Procedure 1. Other tokens are received at line 21: if the token carries a decision, process p_i decides. Note that the stopping of the algorithm is not discussed here. It can easily be added.

4.3.6 Proof of the token-based algorithm

The proofs of the uniform validity and uniform integrity properties are easy and omitted. The proofs of the uniform agreement and termination properties are in the appendix A.2.

5. TOKEN-BASED ATOMIC BROADCAST ALGORITHMS

In this section we show how to transform the token-based consensus algorithm into an atomic broadcast algorithm. Note that we could have presented the atomic broadcast algorithm directly. However, since the consensus algorithm is simpler than the atomic broadcast algorithm, we believe that a two-step presentation makes it easier to understand the atomic broadcast algorithm.

Note also that it is well known how to solve atomic broadcast by reduction to consensus [3]. However, the reduction, which transforms atomic broadcast into a sequence of consensus, yields an inefficient algorithm here. The reduction would lead to multiple instances of consensus, with one token per consensus instance. We want a single token to "glue" the various instances of consensus.

To be correct, the atomic broadcast algorithm requires the failure detector $\mathcal{R}$, a number of processes $n \geq f(f + 1) + 1$, and a vote threshold at $f + 1$ in order to decide, as was the case in the consensus algorithm above.

[7] To avoid complicated notation, we implicitly assume that, for process p_i, waiting a token for $round_i$ means either (1) waiting a token from p_j, $j < i$, with $token.round = round_i$, or (2) waiting a token from p_j, $j > i$, with $token.round = round_i - 1$.

5.1 Overview

In the token-based atomic broadcast algorithm, the token transports (i) sets of messages and (ii) sequences of messages. More precisely, the token carries the following information: *(round, proposalSeq, votes, adeliv, nextSet)*. Messages in the sequence *proposalSeq* are delivered as soon as a sufficient number of consecutive "votes" have been collected. The field *adeliv* is the sequence of all messages adelivered that the token is aware of (in the delivery order). When a process receives the token, it can therefore, if needed, catch up with the message deliveries performed by other processes.

Finally, while the token accumulates votes for *proposalSeq*, it simultaneously collects in *nextSet* the messages broadcast atomically (messages m such that $abroadcast(m)$ has been executed). The set *nextSet* grows as the token circulates. Whenever messages in *proposalSeq* can be delivered, *nextSet* is used as the "proposals" for the next decision.

Procedure 6 Atomic Broadcast: Initialization

1: $\forall p_i,\ i \in [0, n-1]$:
2: $abroadcast_i \leftarrow \emptyset$; $adeliv_i \leftarrow \epsilon$: $round_i \leftarrow 0$

3: p_0 : {*send token*}
4: send(0, $abroadcast_0$, 1, ϵ, $abroadcast_0$) to $\{p_1,..,p_{f+1}\}$
5:
6: $\forall p_i,\ i \in [n-f, n-1]$: {*send "dummy" token*}
7: $send(-1, \emptyset, 0, \epsilon, \emptyset)$ to $\{p_1,\ldots,p_{i+f+1}\}$

Procedure 7 Atomic Broadcast: $abroadcast$ and $adeliver$ (code of p_i)

1: To execute $abroadcast(m)$:
2: $abroadcast_i \leftarrow abroadcast_i \cup \{m\}$

3: To execute $delivery(seq)$:
4: adeliver messages in seq not in $adeliv_i$
5: $adeliv_i \leftarrow adeliv_i \oplus seq$
6: $abroadcast_i \leftarrow abroadcast_i \setminus adeliv_i$

5.2 Details

Each process p_i manages the following data structures (see Procedure 6): $round_i$ (the current round number), $abroadcast_i$ (the set of all messages that have been abroadcast by p_i or another process, and not yet ordered), and $adeliv_i$ (the sequence of messages adelivered by p_i). The algorithm is decomposed into several procedures.

Procedure 6 is the initialization procedure (ϵ denotes the empty sequence).

Procedure 8 Atomic broadcast: token handling by p_i

1: **loop**
2: token $\leftarrow$ receive-token($round_i$) {*see Procedure 1*}

3: $abroadcast_i \leftarrow abroadcast_i \cup token.proposalSeq \cup token.nextSet$

4: **if** $|token.adeliv| < |adeliv_i|$ **then** {p_i *more up to date than the token*}
5: $token.proposalSeq \leftarrow \emptyset$
6: **else** {$|adeliv_i| \leq |token.adeliv|$}
7: $delivery(token.adeliv)$
8: **if** (token received from p_{i-1}) and ($token.proposalSeq \neq \emptyset$) **then**
9: $votes_i \leftarrow token.votes + 1$
10: **else**
11: $votes_i \leftarrow 1$
12: **end if**

13: **if** ($votes_i \geq f + 1$) **then**
14: $delivery(token.proposalSeq)$
15: $token.proposalSeq \leftarrow \emptyset$
16: **end if**
17: **end if**

18: **if** $token.proposalSeq = \emptyset$ **then** {*new proposal can be made...*}
19: $token.proposalSeq \leftarrow abroadcast_i$ {*add new "proposals"*}
20: $votes_i = 1$
21: **end if**
22: token $\leftarrow (round_i, token.proposalSeq, votes_i, adeliv_i, abroadcast_i)$
23: send token to $\{p_{i+1}..p_{i+f+1}\}$
24: $round_i \leftarrow round_i + 1$
25: **end loop**

26: **upon** reception of token s.t.
$token.round < round_i$ **do**
27: **if** $|token.adeliv| > |adeliv_i|$ **then** {*the token has "new" information*}
28: $delivery(token.adeliv)$
29: **end if**

30: $abroadcast_i \leftarrow abroadcast_i \cup token.nextSet$
31: **end upon**

Procedure 7 describes the *abroadcast* and *adelivery* of messages: $delivery(seq)$ is called by Procedure 8. The operator $\oplus$ at line 5 of Procedure 7 is the sequence concatenation operator ($seq_1 \oplus seq_2$ is the sequence of elements in seq_1 concatenated with the sequence of elements in seq_2 that are not in seq_1).

Procedure 8 describes the token-handling. Lines 4 to 17 of Procedure 8 correspond to lines 6-16 of the consensus algorithm (Procedure 5). Procedure $delivery()$ is called to deliver messages (line 14). When this happens, a new sequence of messages can be proposed for delivery. This is done at lines 18 to 21. Finally, lines 26-31 handle

reception of other tokens. This is needed for Uniform Agreement and Validity when $f > 1$. Lines 27-29 are for Uniform Agreement (they play the same role as lines 22-25 of Procedure 5). Line 30 is for Validity (consider $f = 2$, p_i correct and p_{i+1} faulty; without line 30, process p_{i+2} might, in all rounds, receive the token only from p_{i-1}; if this happens, messages *abroadcast* by p_i would never be *adelivered*).

The proof of the algorithm can be derived from the proof of the token-based consensus algorithm.

5.3 Optimization

In our algorithm, the token carries whole messages, rather than only message identifiers. This solution is certainly inefficient. The algorithm can be optimized so that only the message identifiers are included in the token. This can be addressed by adapting techniques presented in other token-based atomic broadcast algorithms, e.g., [4], [13], and is thus not discussed further.

The optimization above reduces the size of the token but does not prevent it from growing indefinitely. This can be handled as follows. Consider a process p that receives the token with the sequence s_1 in the field $adeliv$ and later, in a different round, receives the token with a longer sequence s_2 in the same field (s_1 is a subsequence of s_2). When p receives the token with the sequence s_2, the token containing sequence s_1 has been received by at least $f + 1$ processes, i.e., by at least one correct process. The sequence s_1 can thus be removed from the token. In nice runs (no failures, no suspicions), this means that a process that delivers new messages in round i (thus increasing the size of the $adeliv$ sequence in the token) then removes those messages from the token in round $i + 1$.

The circulation of the token can also be optimized. If all processes are correct, each process actually only needs to send the token to its immediate successor. So, by default each process p_i only sends the token to p_{i+1}. This approach requires that if process p_i suspects its predecessor p_{i-1}, it must send a message to its $f+1$ predecessors,[8] requesting the token. A process, upon receiving such a message, sends the token to p_i. If all processes are correct, this optimization requires only a single copy of the token to be sent by each token-holder instead of $f + 1$ copies, thus reducing the network contention due to the token circulation by a factor $f + 1$.

6. Simulation Results

In this section we compare the performance of our new atomic broadcast algorithm with the Chandra-Toueg algorithm, in which atomic broadcast is solved by reduction to consensus [3]. The Chandra-Toueg algorithm does not use failure detectors directly, but relies solely on consensus (which in turn relies on failure detectors).[9] For consensus, we consider two different algorithms: (1) the Chandra-Toueg consensus algorithm (CT), based on a centralized communication schema [3], and (2) the Mostéfaoui-Raynal consensus algorithm (MR), based on a

[8] Actually, the message does not need to be sent by p_i to p_{i-1}.

[9] This allows us to compare two different atomic broadcast algorithms, both using failure detectors (directly, as in the token-based algorithm, or indirectly, as in the reduction to consensus algorithm, where consensus uses failure detectors).

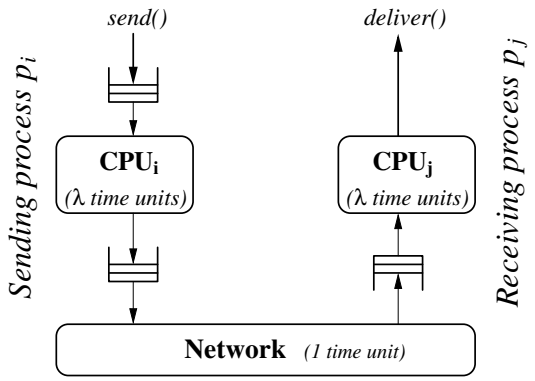


Fig. 1. Sending a message over the Neko simulated network

decentralized communication schema [16]. The two algorithms use the failure detector $\Diamond\mathcal{S}$ and require $f < n/2$. The comparison is done by simulation.

6.1 Simulation model and parameters:

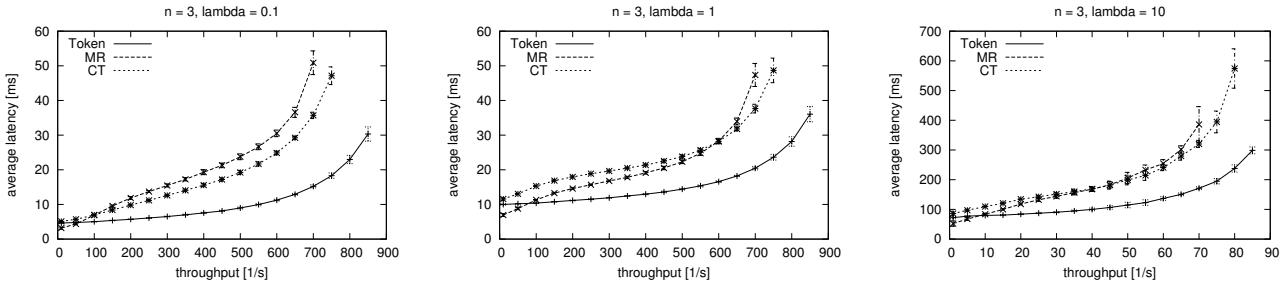
The results have been obtained using the Neko simulation and prototyping framework [18]. Using this framework, the same (Java) implementation of a protocol can be used in a simulated environment and on a real network. The message transmission has been modeled as in [19] and [17].

Both the network and the hosts can be a bottleneck. Each CPU (for sending and receiving messages) and the network are modeled as resources that need to be acquired, used, and finally released. A message m transmitted from process p_i to process p_j (i) first uses the CPU of p_i (with a cost of λ), (ii) then the network (with a cost of 1), and (iii) finally the CPU of p_j (with a cost of λ), as shown in Figure 1. The parameter λ ($\lambda \geq 0$) models the relative speed of processing a message on a host compared to transmitting it over the network: $\lambda = 1$ indicates that CPU processing and transmitting over the network have the same cost, $\lambda > 1$ indicate that CPU processing is expensive compared to transmitting over the network, $\lambda < 1$ indicates that transmitting over the network is expensive compared to CPU processing. We used three representative values $\{0.1, 1, 10\}$ for λ and simulated the algorithms on a multicast network.

6.2 Performance Metric : Latency versus Throughput

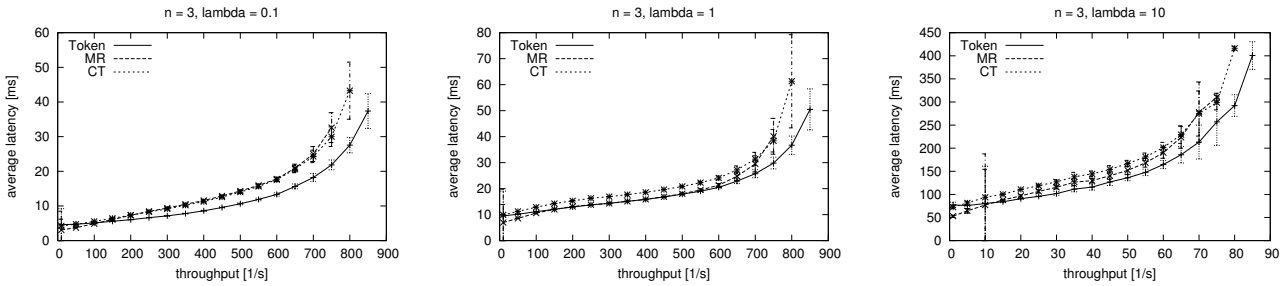
We evaluated the performance of the algorithms with four types of faultloads, as in [19]: *normal-steady* (no failures, no suspicions), *crash-steady* (one or two failures occur before the start of the run, no wrong suspicions), *crash-transient* (failures are injected during the run and detected after a detection time T_D, the performance is measured during the period of instability that follows a crash) and *suspicion-steady* (no failures, but wrong suspicions of average duration T_M with an average recurrence time of T_{MR}).

All of these tests were run with two system settings: (1) $f = 1$: one tolerated failure ($n = 3$ processes for



(a) $\lambda = 0.1$ (b) $\lambda = 1$ (c) $\lambda = 10$

Fig. 2. Latency vs. throughput with a *normal-steady* faultload, $n = 3$ correct processes



(a) $\lambda = 0.1$ (b) $\lambda = 1$ (c) $\lambda = 10$

Fig. 3. Latency vs. throughput with a *crash-steady* faultload, one crashed process (in a group of $n = 3$ processes)

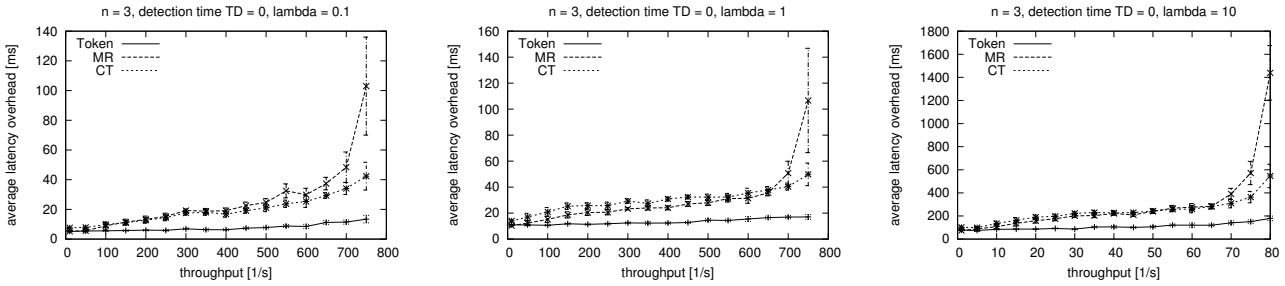
CT, *MR* and *Token*) and (2) $f = 2$: two tolerated failures ($n = 5$ processes for *CT* and *MR*, compared to $n = 7$ processes for *Token*).[10] We use a simple symmetric workload: all processes send atomic broadcasts at the same rate, and the overall rate is called *throughput*. The performance metric for the algorithms is *latency*, defined as the *average* (over all correct processes) of the elapsed time between sending a message m and the delivery of m.

A selection of the results are shown in Figures 2 to 9. The complete simulation results can be found in Appendix B. The graphs give the *latency* as a function of the overall throughput. We set the time unit of the network simulation model to 1 ms, to make sure that the reader is not distracted by an unfamiliar presentation of time/frequency values (one that refers to time units). Any other value could have been used. The 95% confidence interval is shown for each point in the graphs.

6.3 One tolerated failure ($f = 1$)

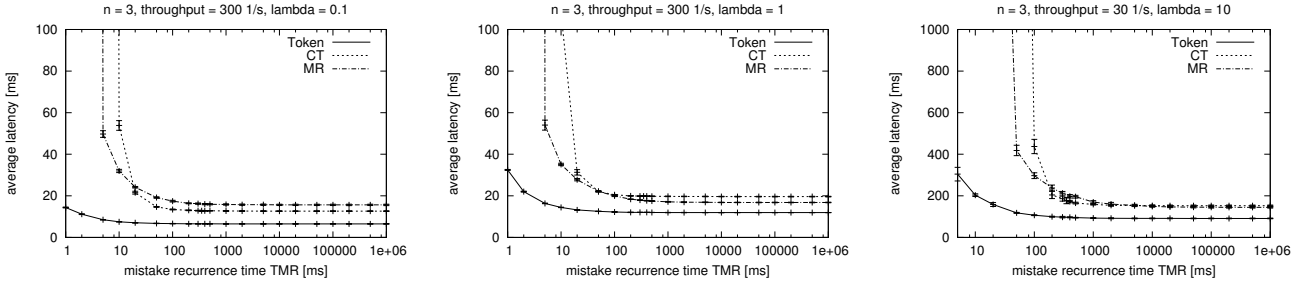
In the case $f = 1$, all algorithms need a system with $n = 3$ processes to guarantee liveness. In such a setting, and with a *normal-steady* faultload (i.e. no failures, no wrong suspicions), the token-based algorithm needs one

[10]The number of processes might seem small, but is adequate to implement scalable atomic broadcast algorithms. Indeed, in a system with a large amount of processes, there is typically a small kernel of "servers" that order the messages and then broadcast them to all other processes. Thus, only the processes in the kernel actually execute the ordering algorithm. For the sake of efficiency, the set of processes included in the kernel should be small. It is therefore reasonable to compare the performance of atomic broadcast algorithms in such a setting.



(a) $\lambda = 0.1$ (b) $\lambda = 1$ (c) $\lambda = 10$

Fig. 4. Latency overhead vs. throughput with a *crash-transient* faultload, one crash (in a group of $n = 3$ processes), detection time $T_D = 0ms$



(a) $\lambda = 0.1$ (b) $\lambda = 1$ (c) $\lambda = 10$

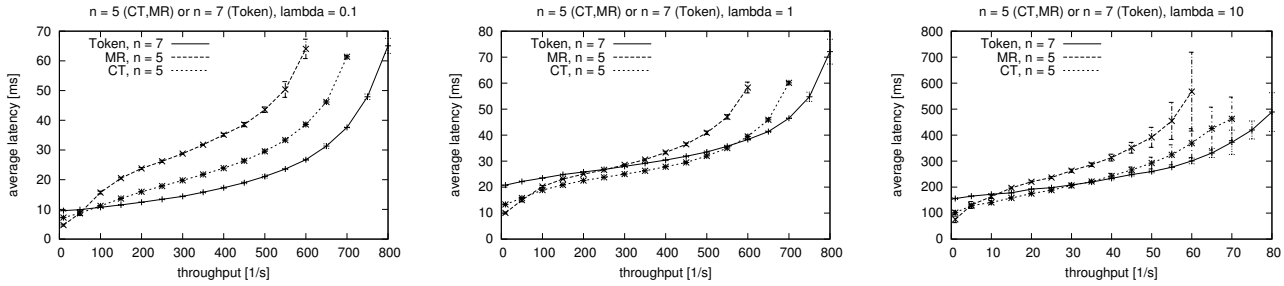
Fig. 5. Latency vs. mistake recurrence time T_{MR} with a *suspicion-steady* faultload $n = 3$ processes, mistake duration $T_M = 0ms$, throughput of 300 ($\lambda = 0.1, 1$) or 30 ($\lambda = 10$) broadcasts per second

broadcast message and one point-to-point message (i.e. two communication steps) per decision. The CT consensus algorithm needs $n = 3$ point-to-point messages and 2 broadcast messages, for a total of 3 communication steps. Finally, the MR algorithm needs $2n = 6$ broadcasts, for a total of 2 communication steps. According to this complexity analysis, the token-based algorithm should perform better than the CT and MR algorithms in a system with 3 processes. Figure 2 confirms this analysis in the case of a run without failures: the token-based algorithm achieves lower latencies than both other algorithms for all loads but the lowest.

In the case of one faulty process (*crash-steady* faultload), the performance gap between the token-based algorithm is significantly smaller (Figure 3), probably due to the decrease of the network contention (only two processes try to access the network) which is favorable to the CT and MR algorithms.

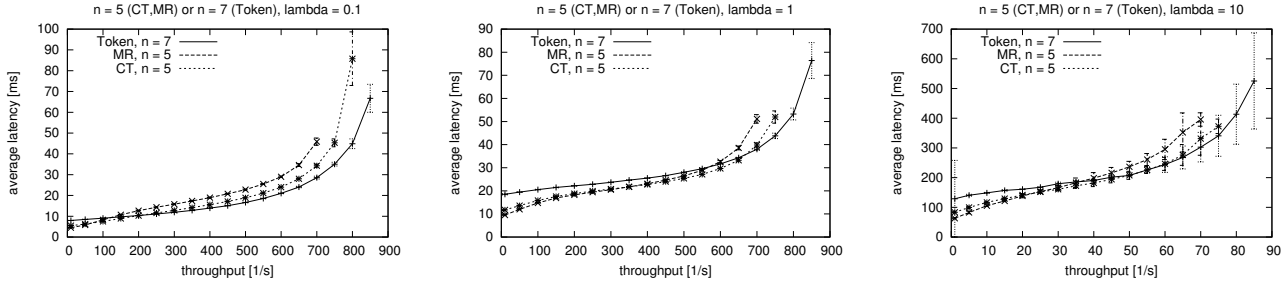
In runs with a *crash-transient* faultload, if the detection is very fast (modelled as detection time $T_D = 0$), the token-based algorithm performs better than both other algorithms, as is shown in Figure 4. With a detection time $T_D = 100$ ms, the token-based algorithm still achieves a slightly lower latency overhead than both other algorithms (the results are shown in Appendix B).

Finally, in runs with wrong suspicions (*suspicion-steady* faultload), the token-based algorithm achieves lower



(a) $\lambda = 0.1$ (b) $\lambda = 1$ (c) $\lambda = 10$

Fig. 6. Latency vs. throughput with a *normal-steady* faultload, $n = 5$ (CT and MR) and $n = 7$ (Token) correct processes



(a) $\lambda = 0.1$ (b) $\lambda = 1$ (c) $\lambda = 10$

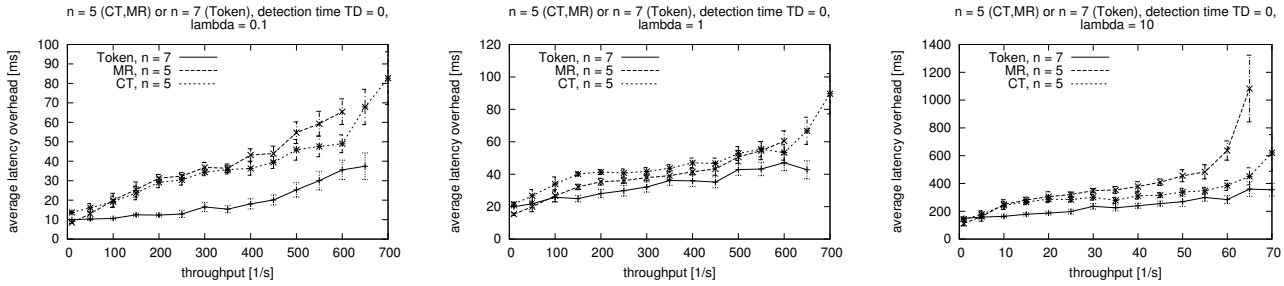
Fig. 7. Latency vs. throughput with a *crash-steady* faultload, two crashed processes (in a group of $n = 5$ (CT and MR) and $n = 7$ (Token) processes)

latencies than the other algorithms, both in the case of frequent failure detector mistakes (small values of T_{MR}) as in the case of less frequent mistakes (Figure 5). The complete simulation results can be found in Appendix B.

6.4 Two tolerated failures ($f = 2$)

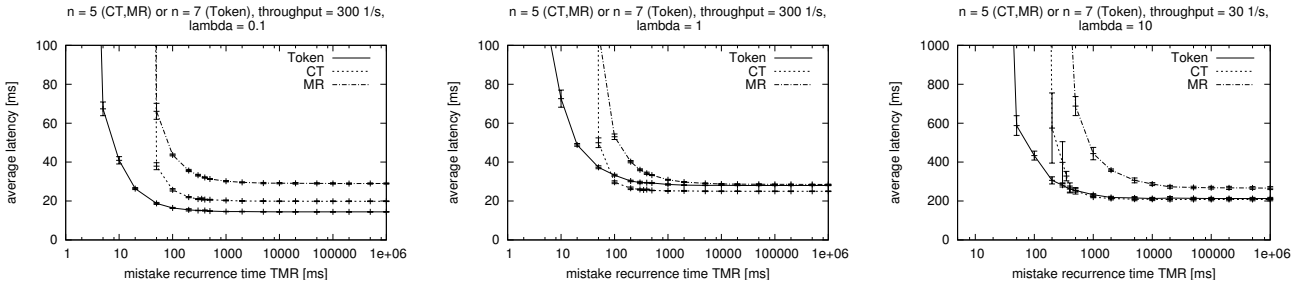
In the case $f = 2$, CT and MR need a system with $n = 5$ processes, whereas the token-based algorithm needs $n = 7$ processes to guarantee liveness. In such a setting, and with a *normal-steady* faultload (i.e. no wrong suspicions), the token-based algorithm needs one broadcast message and between 2 and 3 point-to-point messages (i.e. three to four communication steps). The results for the CT and MR consensus algorithms are as before: $n = 5$ point-to-point messages and two broadcasts for a total of 3 communication steps for CT, $2n = 10$ broadcasts for a total of 2 communication steps for MR.

So, roughly speaking, the token-based algorithm appears better in terms of number of messages, but slightly worse in terms of communication steps. Figures 6 and 7 show that the token-based algorithm performs better than CT and MR in the case of fast processors ($\lambda = 0.1$), except in the case of a very low load. In the case of slower processors ($\lambda = 1, 10$) the token-based algorithm preforms slightly worse than both other algorithms for low throughputs but then achieves better latencies as the throughput increases (when the number of messages, not



(a) $\lambda = 0.1$ (b) $\lambda = 1$ (c) $\lambda = 10$

Fig. 8. Latency vs. throughput with a *crash-transient* faultload, two crashes (in a group of $n = 5$ (CT and MR) and $n = 7$ (Token) processes), detection time $T_D = 0ms$



(a) $\lambda = 0.1$ (b) $\lambda = 1$ (c) $\lambda = 10$

Fig. 9. Latency vs. mistake recurrence time T_{MR} with a *suspicion-steady* faultload, $n = 5$ (CT,MR) and $n = 7$ (Token) processes, mistake duration $T_M = 0ms$, throughput of 300 ($\lambda = 0.1, 1$) or 30 ($\lambda = 10$) broadcasts per second

the communication steps, becomes the dominant factor for the performance of the algorithms).

The performance graphs of the runs with a *crash-transient* faultload (and with $T_D = 0$) show characteristics that are similar to the runs in a failure free system. With this faultload, however, the token-based algorithm achieves lower latencies than the other algorithms even at lower throughput levels, with $\lambda = 1$ and $\lambda = 10$. In the case of very low loads, the token algorithm still performs slightly worse than both other algorithms.

Finally, in runs with a *suspicion-steady* faultload (wrong suspicions), the token-based algorithm performs better than CT and MR as the mistake recurrence time T_{MR} decreases (more frequent wrong suspicions). The complete simulation results can be found in Appendix B.

To wrap up, the simulation results show that the token-based algorithm is a better alternative to other failure detector based algorithms in various system settings, especially in the case $f = 1$ (and except at the lowest loads). In such a case, according to the simulation results, the token-based algorithm achieves lower latencies than both other algorithms, whilst reaching higher throughput levels.

7. RELATED WORK

As was mentioned in Section 1, previous atomic broadcast protocols based on tokens need group membership or an equivalent mechanism. In Chang and Maxemchuk's Reliable Broadcast Protocol [4], and its newer variant [13], an ad-hoc reformation mechanism is called whenever a host fails. Group membership is used explicitly in other atomic broadcast protocols such as Totem [1], the Reliable Multicast Protocol by Whetten et al. [20] (derived from [4]), and in [7].

These atomic broadcast protocols also have different approaches with respect to message broadcasting and delivery. In [4], [20], the *moving sequencer* approach is used : any process can broadcast a message at any time. The token holder then orders the messages that have been broadcast. Other protocols, such as Totem [1] or On-Demand [7] on the other hand use the *privilege based* approach, enabling only the token-holder to broadcast (and simultaneously order) messages.

Finally, the different token-based atomic broadcast protocols deliver messages in different ways. In [7], the token holder issues an "update dissemination message" which effectively contains messages and their global order. A host can deliver a message as soon as it knows that previously ordered messages have been delivered. "Agreed delivery" in the Totem protocol (which corresponds to *adeliver* in the protocol presented in this paper) is also done in a similar way. On the other hand, in the Chang-Maxemchuk atomic broadcast protocol [4], a message is only delivered once $f + 1$ sites have received the message. Finally, the Train protocol presented in [6] transports the ordered messages in a token that is passed among all processes (and is in this respect related to the token-based protocols presented in this paper).

Larrea *et al.* [12] also consider a logical ring of processes, however with a different goal. They use a ring for an efficient implementation of the failure detectors $\Diamond\mathcal{W}$, $\Diamond\mathcal{S}$ and $\Diamond\mathcal{P}$ in a partially synchronous system.

8. CONCLUSION

According to various authors, token-based atomic broadcast algorithms are more efficient in terms of throughput than other atomic broadcast algorithms. The reason is that the token can be used to reduce network contention. However, all published token-based algorithms rely on a group membership service, i.e., none of them use unreliable failure detectors directly. The paper has given the first token-based atomic broadcast algorithms that solely relies on a failure detector, namely the new failure detector called $\mathcal{R}$. Such an algorithm has the advantage of tolerating failures *directly* (i.e., it also tolerates wrong failure suspicions). Algorithms that do not tolerate failures directly, need to rely on a membership service to exclude crashed processes. As a side-effect, these algorithms also exclude correct processes that have been incorrectly suspected. Thus, failure detector based algorithms have advantages over group membership based algorithms, in case of wrong failure suspicions, and possibly also in case of real crashes.

Finally, although token-based atomic broadcast algorithms are usually considered to be efficient only in terms

of throughput, our performance evaluation has shown that for small values of n, our algorithm compares favorably with the Chandra-Toueg atomic broadcast algorithm (using the Chandra-Toueg or Mostéfaoui-Raynal consensus algorithm) in terms of latency as well, at all but the lowest loads. In the future we plan to compare the performance of our new algorithm with token-based algorithms that rely on a membership service, both in nice runs (no crashes, no failure suspicions) and in runs with crashes and wrong failure suspicions.

Acknowledgements. We would like to thank Bernadette Charron-Bost for useful discussions related to failure detectors.

REFERENCES

[1] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Trans. on Computer Systems*, 13(4):311–342, November 1995.

[2] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. on Computer Systems*, 9(3):272–314, Aug. 1991.

[3] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 43(2):225–267, 1996.

[4] J. Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Trans. on Computer Systems*, 2(3):251–273, Aug. 1984.

[5] G. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys*, 4(33):1–43, December 2001.

[6] F. Cristian. Asynchronous atomic broadcast. *IBM Technical Disclosure Bulletin*, 33(9):115–116, 1991.

[7] F. Cristian, S. Mishra, and G. Alvarez. High-performance asynchronous atomic broadcast. *Distributed System Engineering Journal*, 4(2):109–128, June 1997.

[8] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. Technical Report IS–RR–2003–009, Japan Advanced Institute of Science and Technology, Japan, Sept. 2003.

[9] R. Ekwall and A. Schiper. Revisiting token-based atomic broadcast algorithms. Technical Report IC/2003/39, École Polytechnique Fédérale de Lausanne, Switzerland, Feb. 2003.

[10] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of ACM*, 32(2):374–382, Apr. 1985.

[11] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. TR 94-1425, Dept. of Computer Science, Cornell University, Ithaca, NY, USA, May 1994.

[12] M. Larrea, S. Arevalo, and A. Fernandez. Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In *International Symposium on Distributed Computing*, pages 34–48, 1999.

[13] N. F. Maxemchuk and D. H. Shur. An Internet multicast system for the stock market. *ACM Trans. on Computer Systems*, 19(3):384–412, August 2001.

[14] S. Mena, A. Schiper, and P. Wojciechowski. A step towards a new generation of group communication systems. In *Proceedings of Int'l Middleware Conference*, pages 414–432. Springer, June 2003.

[15] S. Mishra and S. M. Kuntur. Newsmonger: A technique to improve the performance of atomic broadcast protocols. *Journal of Systems and Software*, 55(2):167–183, Dec. 2000.

[16] A. Mostéfaoui and M. Raynal. Solving consensus using Chandra-Toueg's unreliable failure detectors: A general quorum-based approach. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC)*, number 1693 in Lecture Notes in Computer Science, pages 49–63, Bratislava, Slovak Republic, Sept. 1999. Springer-Verlag.

[17] P. Urbán, X. Défago, and A. Schiper. Contention-aware metrics for distributed algorithms: Comparison of atomic broadcast algorithms. In *Proc. 9th IEEE Int'l Conf. on Computer Communications and Networks (IC3N 2000)*, pages 582–589, Oct. 2000.

[18] P. Urbán, X. Défago, and A. Schiper. Neko: A single environment to simulate and prototype distributed algorithms. *Journal of Information Science and Engineering*, 18(6), Nov. 2002. To appear.

[19] P. Urbán, I. Shnayderman, and A. Schiper. Comparison of failure detectors and group membership: Performance study of two atomic broadcast algorithms. In *Proc. of the Int'l Conf. on Dependable Systems and Networks (DSN)*, pages 645–654, June 2003.

[20] B. Whetten, T. Montgomery, and S. Kaplan. A high performance totally ordered multicast protocol. In Springer-Verlag, editor, *Theory and Practice in Distributed Systems*, number 938 in Lecture Notes in Computer Science, pages 33–57, Dagstuhl Castle, Germany, Sept. 1994.

APPENDIX A
PROOFS

A.1 Proof of lemma 2 (correctness of the $\mathcal{R}$ to $\Diamond\mathcal{S}$ transformation)

(i) Consider t such that after t all faulty processes have crashed and each correct process p_i has accurate information about its predecessor p_{i-1}. It is easy to see that there is a time $t' > t$ such that after t' all correct processes agree on the same set $correct_i$. Let us denote this set by $correct(t')$.

(ii) The condition $n \geq f(f+1)+1$ guarantees that the set $correct(t')$ contains a sequence of f consecutive processes. Consider the following sequence of processes: 1 faulty, f correct, 1 faulty, f correct, etc. If we repeat the pattern f times, we have f faulty processes in a set of $f(f+1)$ processes. If we add one correct process to the set of $f(f+1)$ processes, there is necessarily a sequence of $f+1$ correct processes. With a sequence of $f+1$ correct processes, there is a sequence of f consecutive processes in $correct(t')$.

(iii) In the case $correct(t') = \Pi$, p_0 and p_1 are trivially correct.

(iv) In the case $correct(t') \neq \Pi$, consider the sequence of $f+1$ processes $p_k, \ldots, p_{k+f}$. Since there are at most f faulty processes, at least one process p_l in $p_k, \ldots, p_{k+f}$ is correct. If $p_l = p_k$, we are done. Otherwise, if p_l is correct, p_{l-1} is correct as well, since the failure detector of p_l is accurate after t' and does not suspect p_{l-1}. By the same argument, if p_{l-1} is correct, p_{i-2} is correct. By repeating the same argument at most $f-1$ times, we have that p_k is correct.

(v) In the case $correct(t') \neq \Pi$, we prove now that p_{k+1} is correct. Since p_k is correct and p_{k-1} is not in $correct(t')$ (by the selection rule of p_k and p_{k+1}), p_{k-1} is faulty. Thus, there are at most $f-1$ faulty processes in the sequence of f processes p_{k+1},...,p_{k+f}. In the special case $f = 1$ ($\{p_{k+1}, \ldots, p_{k+f-1}\} = \emptyset$), all processes in p_{k+1},...,p_{k+f} are correct. In the case $f > 1$, there is a non-empty sequence p_{k+1},...,p_{k+f-1} in $correct(t')$. Furthermore, there are at most $f-1$ faulty processes among the f processes p_{k+1},...,p_{k+f}. By the same argument used to show that p_k is correct, we can show that p_{k+1} is correct. □

A.2 Proof of the token-accumulation algorithm

A sketch of the proof of the uniform agreement and termination properties of the token-accumulation consensus algorithm are presented in the following paragraphs.

a) Uniform Agreement: Let p_i be the first process to decide (say at time t), and let v be the decision value. By line 13 of Procedure 5, we have $votes_i \geq f+1$. Votes are reset for each gap. So, $votes_i \geq f+1$ ensures that at time t, all processes $p_j \in \{p_{i-1}, \ldots, p_{i-f}\}$, have $p_j.estimate = v$. Any process p_k, successor of p_i in the ring, receives the token from one of the processes $p_i, \ldots, p_{i-f}$. Since all these processes have their estimate equal to v, the token received by p_k necessarily carries the estimate v. So after t, the only value carried by the token is v, i.e., any process that decides will decide v. □

b) Termination: Assume at most f faulty processes and the failure detector $\mathcal{R}$. We show that, if $n \geq f(f+1)+1$, then every correct process eventually decides.

First it is easy to see that the token circulation never stops: if p_i is a correct process that does not have the token at time t, then there exists some time $t' > t$ such that p_i receives the token at time t'. This follows from (1) the fact that the token is sent by a process to its $f+1$ successors, (2) the *receive token* procedure (Procedure 1), and (3) the completeness property of $\mathcal{R}$ (which ensures that if p_i waits for the token from p_{i-1} and p_{i-1} has crashed, then p_i eventually suspects p_{i-1} and accepts the token from any of its $f+1$ predecessors).

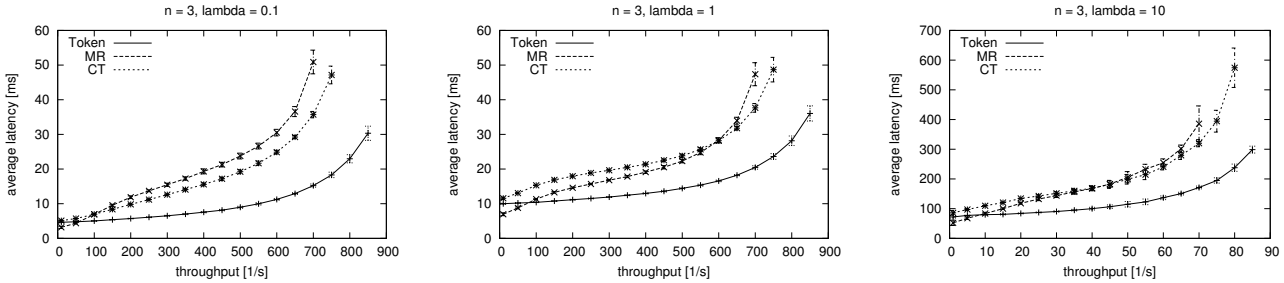
The second step is to show that at least one correct process eventually decides. Assume the failure detector $\mathcal{R}$, and let t be such that after t no correct process p_i is suspected by its immediate correct successor p_{i+1}. Since we have $n \geq f(f+1)+1$ there is a sequence of $f+1$ correct processes in the ring (see Section A.1). Let $p_i \ldots p_{i+f}$ be this sequence. After t, processes $p_{i+1} \ldots p_{i+f}$ only accept the token from their immediate predecessor. Thus, after t, the token sent by p_i is received by p_{i+1}, the token sent by p_{i+1} is received by p_{i+2}, and so forth until the token sent by p_{i+f-1} is received by p_{i+f}. Once p_{i+f} has executed line 9 of Procedure 5, we have $votes_i \geq f+1$. Consequently, p_{i+f} decides.

Finally, if one correct process p_k decides, and sends the token with the decision to its $f+1$ successors, the first correct successor of p_k, by line 21 or line 2, eventually receives the token with the decision and decides (if it has not yet done so). By a simple induction, every correct process eventually also decides. □

APPENDIX B

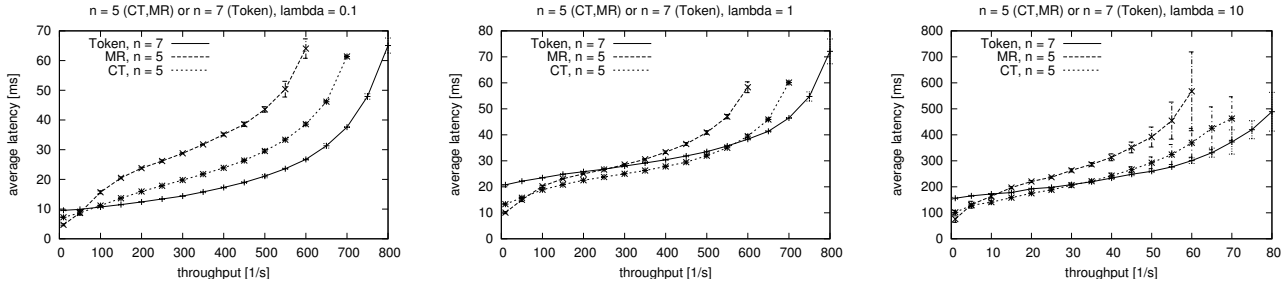
COMPLETE SIMULATION RESULTS

B.1 Normal-steady faultload: no failures, no suspicions



(a) $\lambda = 0.1$ (b) $\lambda = 1$ (c) $\lambda = 10$

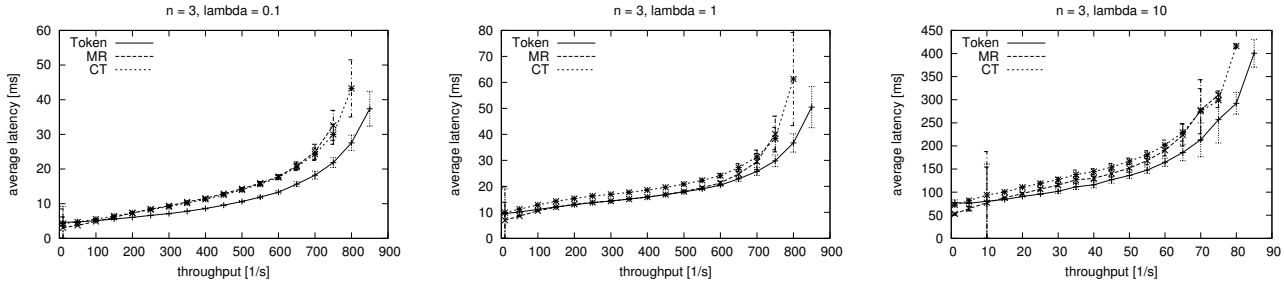
Fig. 10. Latency vs. throughput with the normal-steady faultload, $n = 3$ processes



(a) $\lambda = 0.1$ (b) $\lambda = 1$ (c) $\lambda = 10$

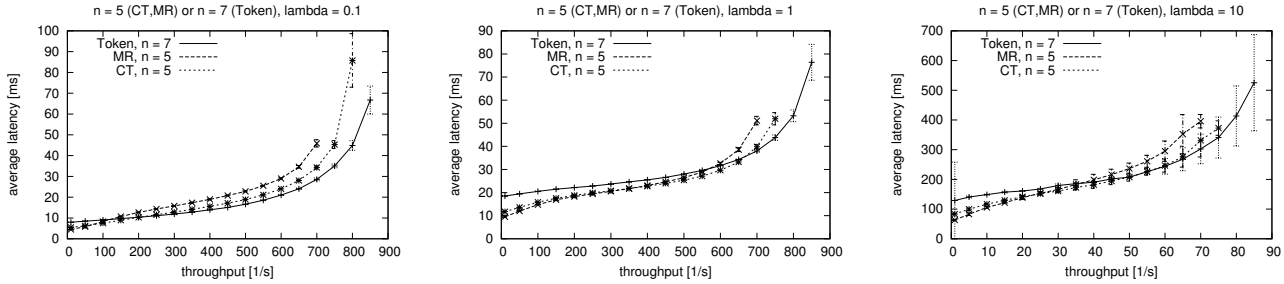
Fig. 11. Latency vs. throughput with the normal-steady faultload, $n = 5$ (CT and MR) and $n = 7$ (Token) processes

B.2 Steady-crash faultload : one or two crashes before the runs, no wrong suspicions



(a) $\lambda = 0.1$ (b) $\lambda = 1$ (c) $\lambda = 10$

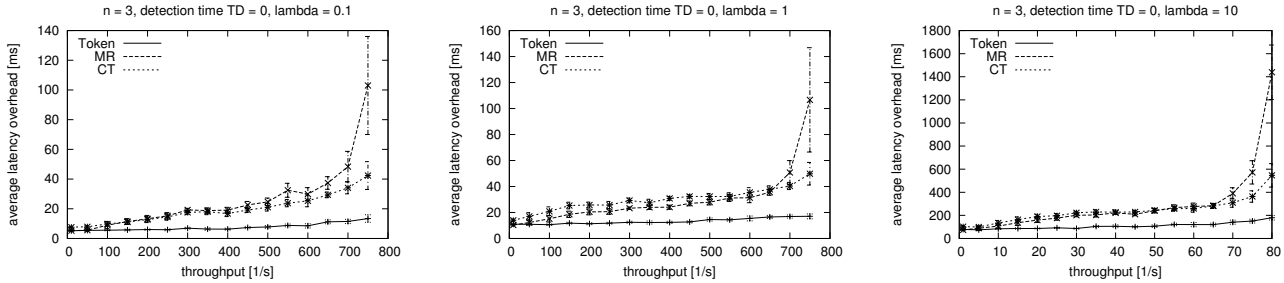
Fig. 12. Latency vs. throughput with the steady-crash faultload, one crash, $n = 3$ processes



(a) $\lambda = 0.1$ (b) $\lambda = 1$ (c) $\lambda = 10$

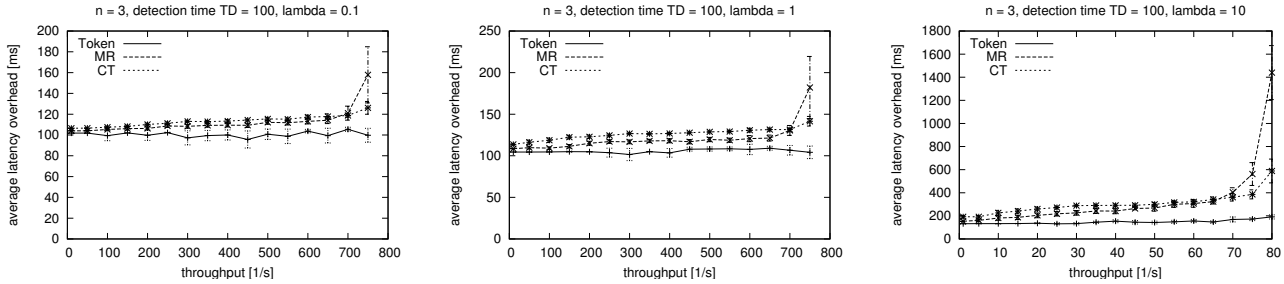
Fig. 13. Latency vs. throughput with the crash-steady faultload, two crashes, $n = 5$ (CT and MR) and $n = 7$ (Token) processes

B.3 Crash-transient faultload : one or two crashes injected during the runs



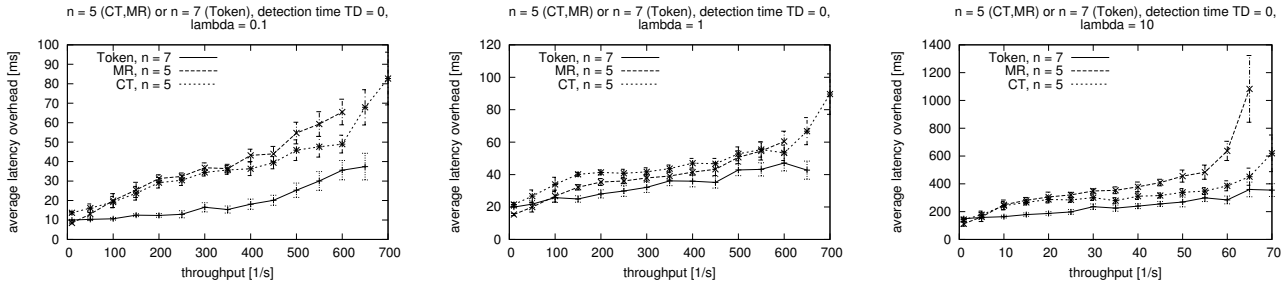
(a) $\lambda = 0.1$ (b) $\lambda = 1$ (c) $\lambda = 10$

Fig. 14. Latency overhead vs. throughput with the crash-transient faultload, one crash (detection time $T_D = 0ms$), $n = 3$ processes



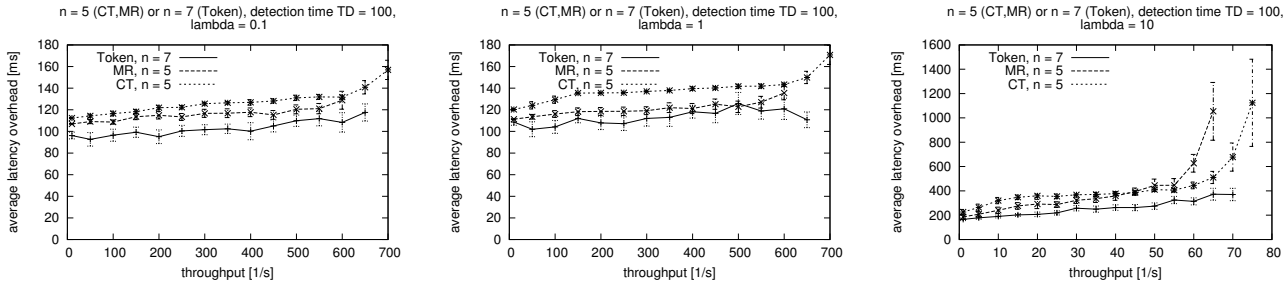
(a) $\lambda = 0.1$ (b) $\lambda = 1$ (c) $\lambda = 10$

Fig. 15. Latency overhead vs. throughput with the crash-transient faultload, one crash (detection time $T_D = 100ms$), $n = 3$ processes



(a) $\lambda = 0.1$ (b) $\lambda = 1$ (c) $\lambda = 10$

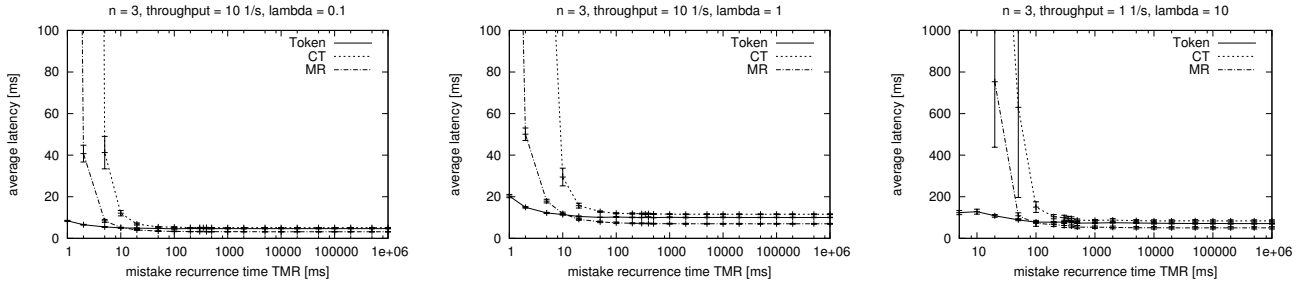
Fig. 16. Latency overhead vs. throughput with the crash-transient faultload, two crashes (detection time $T_D = 0ms$), $n = 5$ (CT and MR) and $n = 7$ (Token) processes



(a) $\lambda = 0.1$ (b) $\lambda = 1$ (c) $\lambda = 10$

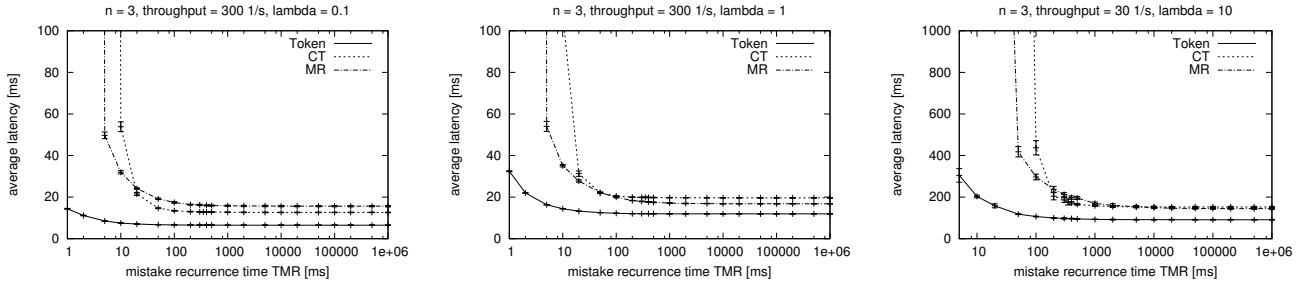
Fig. 17. Latency overhead vs. throughput with the crash-transient faultload, two crashes (detection time $T_D = 100ms$), $n = 5$ (CT and MR) and $n = 7$ (Token) processes

B.4 Suspicion-steady faultload : periodic wrong suspicions that are immediately repaired



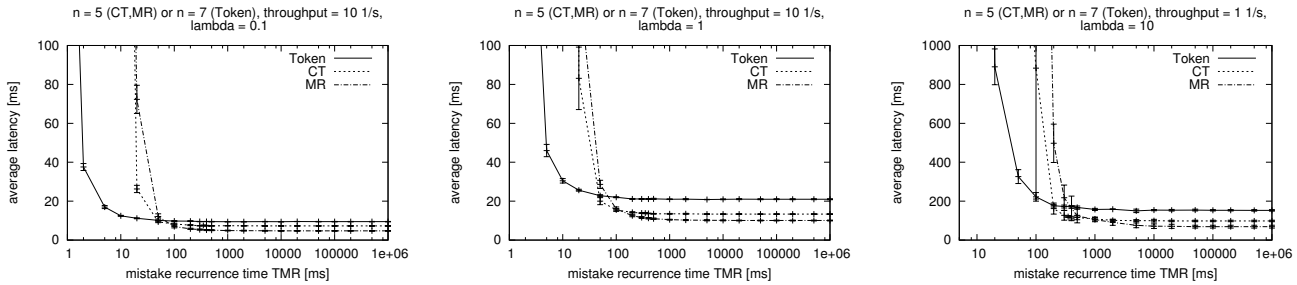
(a) $\lambda = 0.1$ (b) $\lambda = 1$ (c) $\lambda = 10$

Fig. 18. Latency vs. mistake recurrence time (T_{MR}) with the suspicion-steady faultload, $n = 3$ processes, mistake duration $T_M = 0ms$, throughput 10 ($\lambda = 0.1, 1$) and 1 ($\lambda = 10$) broadcasts per second



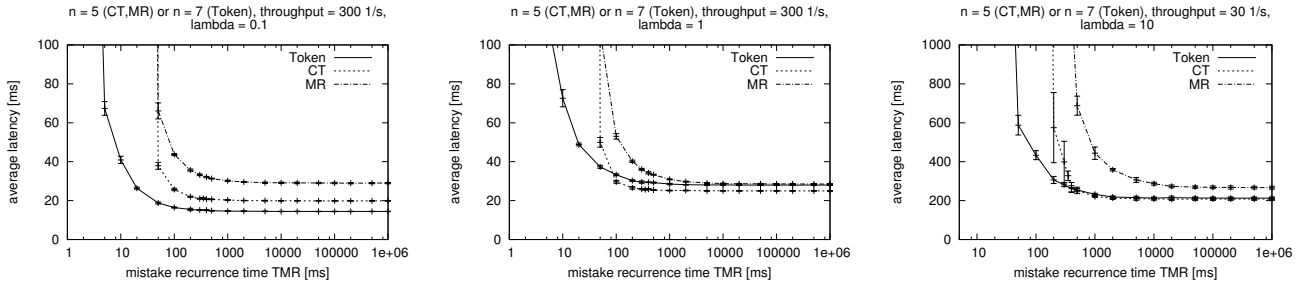
(a) $\lambda = 0.1$ (b) $\lambda = 1$ (c) $\lambda = 10$

Fig. 19. Latency vs. mistake recurrence time (T_{MR}) with the suspicion-steady faultload, $n = 3$ processes, mistake duration $T_M = 0ms$, throughput 300 ($\lambda = 0.1, 1$) and 30 ($\lambda = 10$) broadcasts per second



(a) $\lambda = 0.1$ (b) $\lambda = 1$ (c) $\lambda = 10$

Fig. 20. Latency vs. mistake recurrence time (T_{MR}) with the suspicion-steady faultload, $n = 5$ (CT,MR) and $n = 7$ (Token) processes, mistake duration $T_M = 0ms$, throughput 10 ($\lambda = 0.1, 1$) and 1 ($\lambda = 10$) broadcasts per second



(a) $\lambda = 0.1$ (b) $\lambda = 1$ (c) $\lambda = 10$

Fig. 21. Latency vs. mistake recurrence time (T_{MR}) with the suspicion-steady faultload, $n = 5$ (CT,MR) and $n = 7$ (Token) processes, mistake duration $T_M = 0ms$, throughput 300 ($\lambda = 0.1, 1$) and 30 ($\lambda = 10$) broadcasts per second